2022212153 陈祥烨 计科22-2班 第六章作业

**第二题**

**（1）**

#include<iostream>

using namespace std;

const int v = 0;

const int MAX = 5;

void print(int A[][MAX])

{

for (int i = 0; i < MAX; i++)

{

for (int j = 0; j < MAX; j++)

{

cout << A[i][j] << "\t";

}

cout << endl;

}

}

void print(int A[MAX\*MAX])

{

for (int i = 0; i < MAX \* MAX ; i++)

{

cout << A[i] << "\t";

if (i % 10 == 9)cout << endl;

}

}

int main()

{

int A[MAX][MAX]{0};

int value = 1;

//初始化

for (int i = 0; i < MAX; i++)

{

for (int j = i; j < MAX; j++)

{

A[i][j] = v;

}

for (int j = 0; j < i; j++)

{

A[i][j] = value++;

}

}

//打印A

cout << "A: " << endl;

print(A);

//A转B

int B[MAX \* MAX];

int n = 0;

for (int i = 0; i < MAX; i++)

{

for (int j = 0; j <= i; j++)

{

B[n++] = A[i][j];

}

}

for (; n < MAX \* MAX; n++)

B[n] = v;

//打印B

cout << "B: " << endl;

print(B);

return 0;

}



总结：注意for循环的结束条件。

**（2）**

#include<iostream>

using namespace std;

const int v = 0;

const int MAX = 5;

int cum1(int i)

{

if (i == 0)return 0;

return (i + cum1(i-1));

}

int cum2(int i)

{

if (i == 0)return 0;

return (MAX - i + cum2(i - 1));

}

void print(int A[][MAX])

{

for (int i = 0; i < MAX; i++)

{

for (int j = 0; j < MAX; j++)

{

cout << A[i][j] << "\t";

}

cout << endl;

}

}

void print(int A[MAX\*MAX])

{

for (int i = 0; i < MAX \* MAX ; i++)

{

cout << A[i] << "\t";

if (i % 10 == 9)cout << endl;

}

cout << endl;

}

int main()

{

int A[MAX][MAX]{0};

int value = 1;

//初始化

for (int i = 0; i < MAX; i++)

{

for (int j = i; j < MAX; j++)

{

A[i][j] = v;

}

for (int j = 0; j < i; j++)

{

A[i][j] = value++;

}

}

//打印A

cout << "A: " << endl;

print(A);

//A转B

int B[MAX \* MAX];

int n = 0;

for (int i = 0; i < MAX; i++)

{

for (int j = 0; j <= i; j++)

{

B[n++] = A[i][j];

}

}

for (; n < MAX \* MAX; n++)

B[n] = v;

//打印B

cout << "B: " << endl;

print(B);

//求存储值

int i = 0, j = 0;

int number = 0, val = 0; //位置，元素值

while (1)

{

cout << "input i & j(0<=i<MAX,0<=j<MAX(MAX = " << MAX << ")) (如果i>MAX,j>MAX,则退出循环): " << endl;

cin >> i >> j;

if (i >= j && j >= 0 && i < MAX)

{

number = cum1(i) + j + 1;

val = B[number - 1];

}

else if(i >= MAX || j >= MAX)

break;

else

{

number = cum1(MAX) + cum2(i) + (j - i);

val = v;

}

cout << "A组中的 (" << i << "," << j << ") 元素位于B组中第 " << number << " 位，大小为" << val << endl;

}

return 0;

}



总结：加强对数组行列特性的理解

**第三题**

add.cpp

#include"triple list.h"

int main()

{

//初始化

int a[MAX][MAX] = { 0 };

for (int i = 0; i < MAX; i++)

for (int j = 0; j < MAX; j++)

if (i == j)a[i][j] = 1;

int b[MAX][MAX] = { 0 };

for (int i = 0; i < MAX; i++)

for (int j = 0; j < MAX; j++)

if (i + j == 4)b[i][j] = 1;

TripleList triplelist1(a);

TripleList triplelist2(b);

TripleList triplelist3 = triplelist2 + triplelist1;

cout << "list1:" << endl;

triplelist1.print();

cout << "list2:" << endl;

triplelist2.print();

cout << "list3 = list1 + list2:" << endl;

triplelist3.print();

return 0;

}

triple list.h

#pragma once

#include<iostream>

using namespace std;

const int MAX = 5;

struct Tuple

{

int ti; //行

int tj; //列

int data; //值

Tuple\* next;

};

class TripleList

{

public:

TripleList(); //默认构造函数

TripleList(int A[][MAX]); //三元表构造函数

TripleList(TripleList& tl); //三元表拷贝构造

TripleList(const TripleList& tl) ; //三元表拷贝构造

~TripleList(); //析构函数

int get\_length() const; //获取长度

Tuple\* get\_front() const; //获得头结点

Tuple\* get\_rear() const; //获得尾结点

void append(Tuple\* t); //尾增

void print() const; //打印三元表

TripleList& operator= (const TripleList& tl); //等号运算符重载

TripleList& operator+ (const TripleList& tl); //加号运算符重载

private:

int count;

Tuple \*front,\*rear; //头结点未实例化，尾结点的下一个元素为NULL

};

triple list.cpp

#include"triple list.h"

TripleList::TripleList()

{

Tuple\* tuple = new Tuple;

tuple->next = NULL;

front = tuple;

rear = tuple;

count = 0;

}

TripleList::TripleList(int A[][MAX]) //三元表构造函数

{

Tuple\* tuple = new Tuple;

tuple->next = NULL;

front = tuple;

rear = tuple;

count = 0;

Tuple\* PA = front;

for (int i = 0; i < MAX; i++)

for (int j = 0; j < MAX; j++)

if (A[i][j] != 0)

{

Tuple\* tuple = new Tuple;

tuple->ti = i;

tuple->tj = j;

tuple->data = A[i][j];

tuple->next = NULL;

PA->next = tuple;

PA = tuple;

rear = PA;

count++;

}

}

TripleList::TripleList(TripleList& tl) //三元表拷贝构造

{

Tuple\* tuple = new Tuple;

tuple->next = NULL;

front = tuple;

rear = tuple;

count = 0;

Tuple\* PA = front;

Tuple\* PB = tl.get\_front()->next;

while (PB != NULL)

{

Tuple\* tuple = new Tuple;

tuple->ti = PB->ti;

tuple->tj = PB->tj;

tuple->data = PB->data;

tuple->next = NULL;

PA->next = tuple;

PA = tuple;

rear = PA;

count++;

PB = PB->next;

}

}

TripleList::TripleList(const TripleList& tl) //三元表拷贝构造

{

Tuple\* tuple = new Tuple;

tuple->next = NULL;

front = tuple;

rear = tuple;

count = 0;

Tuple\* PA = front;

Tuple\* PB = tl.get\_front()->next;

while (PB != NULL)

{

Tuple\* tuple = new Tuple;

tuple->ti = PB->ti;

tuple->tj = PB->tj;

tuple->data = PB->data;

tuple->next = NULL;

PA->next = tuple;

PA = tuple;

rear = PA;

count++;

PB = PB->next;

}

}

TripleList::~TripleList() //析构函数

{

Tuple\* temp, \* PA = front;

while (PA != NULL)

{

temp = PA;

PA = PA->next;

delete temp;

}

count = 0;

front = NULL;

rear = NULL;

}

void TripleList::append(Tuple\* t) //尾增

{

Tuple\* tuple = new Tuple;

tuple->ti = t->ti;

tuple->tj = t->tj;

tuple->data = t->data;

tuple->next = NULL;

rear->next = tuple;

rear = tuple;

count++;

}

void TripleList::print() const //打印三元表

{

if (count == 0)

cout << "list is empty!" << endl;

else

{

Tuple\* PA = front->next;

cout << "i\tj\tdata" << endl;

while (PA != NULL)

{

cout << PA->ti << "\t" << PA->tj << "\t" << PA->data << endl;

PA = PA->next;

}

}

}

int TripleList::get\_length() const //获取长度

{

return count;

}

Tuple\* TripleList::get\_front() const //获得头结点

{

return front;

}

Tuple\* TripleList::get\_rear() const //获得尾结点

{

return rear;

}

TripleList& TripleList::operator= (const TripleList& tl) //等号运算符重载

{

//删除原有数据

Tuple\* temp, \* PA = front;

while (PA != NULL)

{

temp = PA;

PA = PA->next;

delete temp;

}

count = 0;

front = NULL;

rear = NULL;

//深拷贝新数据

Tuple\* tuple = new Tuple;

tuple->next = NULL;

front = tuple;

rear = tuple;

count = 0;

Tuple\* NPA = front;

Tuple\* PB = tl.get\_front()->next;

while (PB != NULL)

{

Tuple\* tuple = new Tuple;

tuple->ti = PB->ti;

tuple->tj = PB->tj;

tuple->data = PB->data;

tuple->next = NULL;

NPA->next = tuple;

NPA = tuple;

rear = NPA;

count++;

PB = PB->next;

}

return \*this;

}

TripleList& TripleList::operator+ (const TripleList& tl)//加号运算符重载

{

TripleList\* C = new TripleList(tl);

Tuple\* PA;

Tuple\* PC;

bool isappend = 1;

for (PA = front->next; PA != NULL; PA = PA->next, isappend = 1)

{

for (PC = C->get\_front()->next; PC != NULL; PC = PC->next)

{

//坐标相等，则值相加；遍历完没有相等，则 C 尾增

if (PC->ti == PA->ti && PC->tj == PA->tj)

{

PC->data += PA->data;

isappend = 0; //为1 表示矩阵中该位置是0，三元组表需要尾增

}

}

if (isappend == 1)

C->append(PA);

}

return \*C;

}



总结：原理我就不说啦，说一下我在编码时出现的问题：

代码如下（灰色为问题所在）

TripleList& TripleList::ADD(TripleList& tl) //三元表相加并返回为新表

{

TripleList C(tl);

Tuple\* PA;

Tuple\* PC;

bool isappend = 1;

for (PA = front->next; PA != NULL; PA = PA->next, isappend = 1)

{

for (PC = C.get\_front()->next; PC != NULL; PC = PC->next)

{

//坐标相等，则值相加；遍历完没有相等，则 C 尾增

if (PC->ti == PA->ti && PC->tj == PA->tj)

{

PC->data += PA->data;

isappend = 0; //为1 表示矩阵中该位置是0，三元组表需要尾增

}

}

if (isappend == 1)

C.append(PA);

}

return C;

}

TripleList C(tl)；这里的是 C 是临时变量，函数调用完会删除，会执行TripleList析构函数，使得返回的值没有意义。所以需要开辟新空间，还需要写一份const的拷贝构造函数，以保证传入数据的不变性。同时升级一下，普通函数升级为运算符的重载。可以连加的那种。

//……

TripleList triplelist3 = triplelist2 + triplelist1 + triplelist2;

//……



**第六题**

**第七题**

test.cpp

#include"Generalized List.h"

void Test()

{

GList genList("(a,b,(c,d),(e,(f),h))");

genList.Print();

cout << endl;

cout << "Size is :" << genList.Size() << endl;

cout << "Depth is :" << genList.Depth() << endl << endl;

}

int main()

{

Test();

return 0;

}

Generalized List.h

#pragma once

#include<iostream>

using namespace std;

enum GNodeType {HEAD = 0, VALUE = 1, SUB = 2};

struct GListNode

{

GNodeType type; //元素类型

union //数据域

{

char data; //有效值

GListNode\* sublist; //指向指标的指针

};

GListNode\* next; //下一个指针

GListNode(GNodeType type = HEAD, char value = '\0') :type(type), next(NULL)

{

if (type == VALUE)

{

data = value;

}

else if (type == SUB)

{

sublist = NULL;

}

}

};

class GList

{

private:

GListNode\* \_front; //头结点指针

public:

GList(const char\* str) : \_front(NULL)

{

\_CreateGList(\_front,str); //根据指定序列创建广义表

}

~GList(){} //析构函数

public:

void Print(); //打印表 对外接口

int Size(); //获取广义表中值结点数目 对外接口

int Depth(); //获取广义表的最深层次数 对外接口

private:

void \_CreateGList(GListNode\*& link, const char\*& str); //构造函数底层逻辑

bool \_IsValue(const char ch); //判断指定字符是否为值结点所允许的类型

int \_Size(GListNode\* head); //获取广义表中值结点数目 函数实现

int \_Depth(GListNode\* head); //获取广义表的最深层次数 函数实现

void \_Print(GListNode\* link); //打印表 函数实现

};

Generalized List.cpp

#include"Generalized List.h"

void GList::\_CreateGList(GListNode\*& link, const char\*& str)

{

//创建头结点

GListNode\* head = new GListNode(HEAD, NULL);

head->next = NULL;

link = head;

//用于记录当前指针

GListNode\* cur = link;

str++;

while (\*str != '\0')

{

if (\_IsValue(\*str)) //如果当前扫描到的字符是值

{

GListNode\* newnode = new GListNode(VALUE, \*str);

newnode->next = NULL;

cur->next = newnode;

cur = cur->next;

str++;

}

else if (\*str == '(') //如果是'(',则创建子表节点

{

GListNode\* subnode = new GListNode(SUB, NULL);

subnode->next = NULL;

cur->next = subnode;

cur = cur->next;

\_CreateGList(cur->sublist, str); //递归创建子表

}

else if (\*str == ')') //子表创建结束

{

str++;

return;

}

else //无效字符跳过

{

str++;

}

}

}

bool GList::\_IsValue(const char ch)

{

if (ch >= 'a' && ch <= 'z' ||

ch >= 'A' && ch <= 'Z' ||

ch >= '0' && ch <= '(')

return true;

return false;

}

int GList::Size()

{

return \_Size(\_front);

}

int GList::\_Size(GListNode\* head)

{

int size = 0;

GListNode\* cur = head;

while (cur != NULL)

{

if (cur->type == VALUE)

size++;

else if (cur->type == SUB)

size += \_Size(cur->sublist); //遇到子表进行递归

cur = cur->next;

}

return size;

}

int GList::Depth()

{

return \_Depth(\_front);

}

int GList::\_Depth(GListNode\* head)

{

int depth = 1, maxdepth = 1; //maxdepth为目前最大深度

GListNode\* cur = head;

while (cur != NULL)

{

if (cur->type == SUB)

depth += \_Depth(cur->sublist);

if (depth > maxdepth)

{

maxdepth = depth;

depth = 1;

}

cur = cur->next;

}

return maxdepth;

}

void GList::Print()

{

\_Print(\_front);

}

void GList::\_Print(GListNode\* link)

{

GListNode\* cur = link; //遍历游标

while (cur != NULL)

{

if (cur->type == VALUE)

{

cout << cur->data;

if (cur->next != NULL)

cout << ", ";

}

else if (cur->type == HEAD)

cout << "( ";

else if (cur->type == SUB)

{

\_Print(cur->sublist);

if (cur->next != NULL)

cout << ", ";

}

cur = cur->next;

}

cout << " )";

}



总结一：

为什么构造子节点要传进去二级指针？

刚开始我传进一级指针的时候，我发现创建完成后，gl = NULL，为什么？

因为没有一个变量记录gl第一个节点的地址啊，所以最后找不到它了。

这时候就需要一个二级指针，指向该头节点的地址。

总结二：

外部操作自定义数据类型，设置独立对外接口，加强数结构安全性，减少无操作

思考一：

为什么不把深度，值结点长度设为参数？

广义表相比单链表，链队列等深度为1的数据结构而言，插入，删除等操作更为复杂，

对于总数的变化不好统计，并且在复杂程序中插入删除操作次数会远多于求深度的操作

总体而言，求深度单另设一函数，减少运算时间